Energy Delay Product of Memory Scheduler Implementations

A. Emis, *Student, UCLA*

I. First Come First Served Scheduler

I was tasked with implementing a memory scheduler with the intention of minimizing the Energy Delay Product (EDP) of the traces given. I was given a First Come First Served Scheduler as the base implementation. They defined a high water mark and low water mark as 40 and 20 respectively to monitor the number of writes in the write queue. If there were more than 40, the writes would be drained until the write queue length was equal to 20. If the writes were being drained, they would attempt to issue a write command. Using the LL_FOREACH function to traverse through the write queue linked list (starting at the head), the program checks if that write is command issuable. If it is command issuable, then the program would call issue_request_command on that write and the function would return. If writes aren't being drained, they would focus on draining reads instead. Following a similar process as draining writes, the program would use LL_FOREACH (starting with the head node) to traverse through the read queue. For each entry in the read queue, if it is command issuable, the program calls issue_request_command and returns. If no command is issued, the program just returns. This implementation can be a good option if simplicity is a valuable goal. However, it is not a very fair implementation since the oldest commands are always prioritized over the newest ones. This base implementation returns an average EDP of 0.471.

II. Close Page Scheduler

I was also given a Close-Page scheduler that worked similarly to the First Come First Served implementation, with the high and low water marks determining if the program was in drain writes mode or not and traversing the read/write queues for the first issuable command. The main difference is that the First Come First Served doesn't do anything on idle cycles (cycles where no command is issued). This close page scheduler issues precharge instructions to banks that just did a column read or write [1]. Precharging is essentially helping the memory prepare for the next instruction that it has to take care of. When doing a column read or write, it is likely that the row that was just activated will have to be deactivated in order to activate the next row in that column, so if no command was issued already, it is worth issuing that helpful command to prepare for the upcoming instructions. This implementation keeps a data structure to see if a bank is a good candidate for being precharged. If the next command is a COL_WRITE_CMD, and not a ACT_CMD or PRE_CMD, it is a candidate for precharging. If a command hasn't been issued yet, a nested for loop is used to traverse the data structure to find the first bank in that channel that is a candidate for precharge. If the precharge is allowed, issue_precharge_command is called. This implementation has an average EDP of 0.441, which is better than the First Come First Served implementation because idle cycles are still productive.

III. Round Robin Scheduler

I wanted to build off of and improve the given implementations by creating a Round Robin Scheduler. The idea behind this is to make fairness a top priority and make sure no thread is being starved by other greedy threads. By building off of the First Come First Served Scheduler, this meant traversing through the entries in the queue until finding one whose thread id matched the current round robin value. If one was found, the round robin value would be incremented and that command would be issued (if it was valid). If one was not found, this implied the round robin value might have exceeded the number of threads, so the round robin value would be reset. I spent a few hours on my first implementation of this and was confused why it didn't speed up at all. I realized I was checking the channel value instead of the thread id, which is not what I should have been doing for a round robin implementation. I first thought the goal was to improve channel fairness, when it should have been improving thread fairness. When I focused on channel fairness, I could only achieve an EDP of 0.482, which is worse than the baseline. After fixing the round robin comparison, I added a few more options, following the round robin idea in [1]. The program tries to find a command with the correct thread-id that hits an open row. If none is found, look for commands with open row hits, regardless of thread id. If none are found, look for row misses with the correct thread id. If none are found again, just default to the First Come First Served implementation. It took a bit of time for me to realize, but I could determine if a row was open or not [2] by using the following code: dram_state[channel][rd_ptr->dram_addr.rank][rd_ptr->dram_addr.bank].state == ROW_ACTIVE. I was able to get the EDP to a value of 0.463, which was better than the baseline. I combined this current implementation with the close page scheduler implementation to make sure the idle cycles were still productive. This got me my best EDP of 0.437. My implementation outperformed the baseline, but has some downsides, such as requiring extra storage for the close page scheduler and having somewhat "bloated" code that checks a lot of similar things (like if the row is open and if the thread id matches the round robin value) multiple times.

References

1. Chatterjee, N., Balsubramonian, R,. Shevgoor, M., Pugsley, S., Udipi, A., Shafiee, A., Sudan, K., Awasthi, M., Chishti, Z. *"USIMM: the Utah Simulated Memory Module."* https://users.cs.utah.edu/~rajeev/pubs/usimm.pdf

2. Mutlu, O. *"Computer Architecture: Main Memory (Part 1)"* Carnegie Mellon University. https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=seth-740-fall13-module3.5-main-memory-part1.pdf